Httpcore 源码阅读报告

作者：燕澄皓

课程名称：面向对象程序设计

授课教师：王伟

前言

此报告是本人在参加中国科学院大学 2019-2020 学年《面向对象程序设计》课程时为完成开源项目分析大作业而撰写的。

报告撰写的分为三大模块，分别对应于课程进展中三个阶段的要求：

1. 主要功能及交互流程分析
2. 类、接口的设计及关联，面向对象思想体现
3. 设计模式探讨及高级设计意图分析

由于本人能力有限，在阅读的过程中难免出现理解偏差或表达不准的地方。若读者在阅读的过程中发现此报告中存在错误或者疏漏，请通过邮箱 yanchenghao17@mails.ucas.ac.cn 联系我。此外，对于报告如果存在任何疑问或者是想与作者进行进一步的交流，均可通过上述邮箱联系本人，希望能在探讨的过程中与诸位读者一起进步。

一、 Httpcore 主要功能及交互流程简介

1. 什么是 HTTP?

HTTP 全称 Hypertext Transfer Protocol，又名超文本传输协议，是一种为共有的超文本信息传输系统定制的应用层协议，通俗来讲就是交互信息的双方应当遵循的一种规范。HTTP 是一个客户端和服务器端请求和应答的标准(Transmission Control Protocol, TCP)

HTTP 协议是一种无状态的协议，具体来说就是服务器在与客户交互时不会记录历史信息，在服务器的视角里并不知道客户端实际做了什么操作。这是一种基于性能上的考虑，它减轻了服务器的记忆负担并加快了服务器对客户端请求的响应速度。现实中一个服务器往往同时处理着成千上万个客户端的请求，为了使得服务器能够承载这样庞大的负担，HTTP 协议规定在客户端每次发送请求都要重新获得与服务器的链接，而一旦收到服务器的回应就断开本次链接，而不是让服务器一直处于轮询等待的状态。HTTP 协议属于应用层的协议，它规定了客户端应用程序与服务器端进行信息交互的规则与格式，是真正面向事务的协议。我们在互联网上浏览网页、查阅资料等一系列请求都以 HTTP 协议作为底层的重要基础。

HTTP 协议采用的是一套请求/响应的模型，一次完整的交互流程至少包括请求、响应这两大环节。对工作流程作以简单概述：客户端基于自己的需求向服务器端发送一个 HTTP 请求，在请求中声明自己想要访问的资源以及请求的具体动作。服务器在接收到请求之后开始进行处理，解析客户端的请求并依据请求进行相应的动作来访问服务器上的资源，并返回一个状态行给客户端说明此次响应的结果。一次完整的响应往往包括消息协议的版本，服务器信息，成功或者错误的编码以及相应的解释字段，实体元信息以及可选取的实体内容。

2. HTTPCORE 及其主要功能

有了一套完整、规范的协议，下一个问题就是如何对协议进行封装、抽象并利用其进行高效的工作。

HttpComponents 对于底层的 Http 协议进行了良好的封装，为使用者提供了一套功能完善、操作便捷的 API，而无需程序员本身基于底层的 Http 协议来进行操作。

HttpCore 是 HttpComponents 中的一个核心组件，它是基于超文本传输协议的一种实现方式。作为一组底层 Http 传输协议组件，它为客户端以及服务器端的信息交互提供了一组 I/O 接口，用来按照一定的格式规范对 HTTP 消息的表示以及信息传输时的逻辑进行处理。HttpCore 实现了客户端与服务器端信息交互的一致性，保障了信息交互过程中的安全性及高效性。在整个 HttpComponents 中，它是处于核心地位的一个基础模块。

HttpCore 需要负责处理消息（HttpMessage）的表示，它提供了一系列用于创建、管理 HttpMessage 的接口，譬如用于请求的 HttpRequest，用于响应的 HttpResponse 等。实际上 HttpCore 最主要干的事情就是处理好各种对 HttpMessage 的操作，这其中牵扯到诸多的细节。一则 HttpMessage 中包含多个复杂的组成部分，譬如请求行 RequestLine、状态行 StatusLine、消息头 HttpHeaders、消息实体 HttpEntity 等等。消息头中又包含了消息头的名字以及值，而消息头的值又可以包含多个 HttpElement 等等。可以看出的是进行的一系列复杂操作都是为了管理好 HttpMessage，所以要分析好 HttpCore，应该首先着眼于分析 HttpMessage 这个最核心的部分。

二、 类的设计以及关联，面向对象思想体现

在阅读整个 HttpCore 的大致框架之后发现，整个工程的架构基本符合高内聚低耦合的设计原则。代码包含 annotation，concurrent，config，entity，impl，message 等多个包，而这些包之间的关联度都很低，这也就意味着每个包的功能明确，分析也可以相对较为独立地进行；而同一包内地各文件则呈现出较为紧密的关联。项目的设计呈现出结构化层次化的特点，底层的几个基础模块被封装好，被其它更高层次的模块广泛依赖，因而遵循上面的分析，仍旧从底层的 Message 入手分析整个项目。

HttpMessage 实现了 HttpCore 的主要功能，它提供了客户端到服务器的请求消息以及服务器给客户端的响应消息的接口。即 HttpMessage 作为一个接口，提供了对所有消息最顶层的一种抽象。首先我们简要列举一下这个接口所提供的方法：

```java
/**
 * Checks if a certain header is present in this message. Header values are
 * ignored.
 *
 * @param name the header name to check for.
 * @return true if at least one header with this name is present.
 */
boolean containsHeader(String name);

/**
 * Returns all the headers with a specified name of this message. Header values
 * are ignored. Headers are orderd in the sequence they will be sent over a
 * connection.
 *
 * @param name the name of the headers to return.
 * @return the headers whose name property equals {@code name}.
 */
Header[] getHeaders(String name);

/**
 * Returns the first header with a specified name of this message. Header
 * values are ignored. If there is more than one matching header in the
 * message the first element of {@link #getHeaders(String)} is returned.
 * If there is no matching header in the message {@code null} is
 * returned.
 *
 * @param name the name of the header to return.
 * @return the first header whose name property equals {@code name}
 *   or {@code null} if no such header could be found.
 */
Header getFirstHeader(String name);
```

图 1　HttpMessage 方法示例

截图中给处的是 HttpMessage 这个接口中提供的三种方法，containsHeader 判断消息中是否含有消息头，getHeaders 返回与 name 相匹配的消息头，getFirstHeader 用于返回第一个与 name 相匹配的消息头。除过这三种操作之外还有诸如 addHeader，removeHeader 等等，可以看出 Message 这个接口提供了一系列维护消息头（Header）或处理消息头的方法，我们起码能认识到 Header 是 Message 中一个重要的成分（它的功能以及实现特点在之后进一步分析）。但是我们注意到，HttpMessage 这个接口并没有针对不同的消息类型进行区别，也没有任何实质性的代码，故而 HttpMessage 仅是一种对消息的顶层抽象，只从概念上提供了对一般的消息的处理方法。

HttpRequest 以及 HttpResponse 则分别是请求以及响应消息对 HttpMessage 这个接口的继承，可以看如下截图：

```java
public interface HttpRequest extends HttpMessage {

    /**
     * Returns the request line of this request.
     * @return the request line.
     */
    RequestLine getRequestLine();

}
```

图 2　HttpRequest 接口

上图是 HttpRequest 这一接口，可以看到它只是在继承 HttpMessage 的接口的基础上又提供了获取 RequestLine（请求行）的方法；从这个接口的简单定义我们可以看出，一般的请求有请求行 StatusLine 以及消息头 Header 的信息就足够了，所以没有在这个接口中提供更多的操作方法；但显然有些请求还会包含更复杂的内容，比如含有消息实体等，对于这些更复杂一点的请求 Httpcore 提供了 HttpEntityEnclosingRequest 这一接口，它在继承通用的 HttpRequest 接口的基础上又提供了诸如 getEntity、setEntity 等对消息实体操作。可以看出，对于简单以及复杂两种请求 Httpcore 提供了两个层次的接口，这样的设计使得代码的层次结构更加分明；它们都是 HttpRequest 的接口，但是彼此之间又明显的层次关系，这体现了面向对象程序设计中关键的一个理念或方法：从同一类事物中提取共性，再根据不同进行层次化。而与 HttpRequest 相比，HttpResponse 则没有“简单”、“复杂”这个概念上的差别，HttpResponse 给响应消息提供了一致的接口：

```java
public interface HttpResponse extends HttpMessage {

    StatusLine getStatusLine();

    void setStatusLine(StatusLine statusline);
    void setStatusLine(ProtocolVersion ver, int code);
    void setStatusLine(ProtocolVersion ver, int code, String reason);

    void setStatusCode(int code)
        throws IllegalStateException;

    void setReasonPhrase(String reason)
        throws IllegalStateException;

    HttpEntity getEntity();

    void setEntity(HttpEntity entity);

    Locale getLocale();

    void setLocale(Locale loc);

}
```

图 3　HttpResponse 接口

上图是 HttpResponse 接口，它同样继承了 HttpMessage，在此基础上提供了获取

状态行 StatusLine 的方法、三个版本的设置状态行的方法以及其他对 StatusCode、ReasonPhrase、消息实体 Entity 等的操作方法。我们可以通过分析该接口大致了解一个 Response 消息搭建的若干主要操作。

首先指出 HttpResponse 是一个响应的接口，而一则响应消息的种种组成部分是需要用 get 方法从系统的其它部分去尝试获取的。而该接口则提供了对于响应消息按响应格式及需求组装、搭建响应消息的种种方法。我们指出一则响应消息最重要的部分是状态行 StatusLine，而对于不同的响应格式以及需求该接口提供了三种不同的设置 StatusLine 的方法（比如是否有版本协议号、解释行，状态码等等）；此外该接口还提供了获取、编写消息实体 Entity、获取、设置国际化地区等操作。但是尽管分析了这么多，我们仍然要意识到，HttpRequest 以及 HttpResponse 也仅仅是比 HttpMessage 低一层的抽象，它们分别提供了对一般的请求以及响应消息操作的接口，但是方法仍然没有实现。

接下来我们看看这些接口具体的实现。在 http/message 目录下的 AbstractHttpMessage 这一抽象类实现了 HttpMessage 这一接口，对于接口所提供的种种方法进行了 Override 重写，这里我们截取一部分方法的实现进行简要说明：

```
@Override
public boolean containsHeader(final String name) {
    return this.headergroup.containsHeader(name);
}

// non-javadoc, see interface HttpMessage
@Override
public Header[] getHeaders(final String name) {
    return this.headergroup.getHeaders(name);
}

// non-javadoc, see interface HttpMessage
@Override
public Header getFirstHeader(final String name) {
    return this.headergroup.getFirstHeader(name);
}

// non-javadoc, see interface HttpMessage
@Override
public Header getLastHeader(final String name) {
    return this.headergroup.getLastHeader(name);
}
```

图 4　AbstractHttpMessage 方法示例

我们之前所分析的 HttpMessage 提供了对消息头进行种种操作的接口，但在该抽象类对于接口的实现里我们看到，这些接口提供的方法也不是在类里具体实现的，而是“委托”给一个容器对象 headergroup 来实现的，这又是一层封装。从概念上理解，对于消息头的维护是一个消息头的容器 Headergroup 应该负责好的，一个更高层次的消息抽象类只需要从消息头容器中得到这些信息即可。这样做的好处在于使得代码的层次更加清晰，也降低了代码的维护成本。当维护消息头的这些操作的代码需要变更时，不需要更改整个 AbstractHttpMessage 这个明显更高层次的抽象类，只需要更改在 headergroup 这个底层一点的容器类里的具体方法实现即可。

AbstractHttpMessage 是一般消息的一个抽象类，httpcore 对于响应以及请求分别提供了两个基础的实现类 BasicHttpResponse、BasicHttpRequest，它们在继承抽象类 AbstractHttpMessage 的基础上分别又实现了各自专用的接口 HttpResponse 以及 HttpRequest。我认为这一部分的设计十分巧妙，对于层次化的划分以及模块化的封装做的十分到位。对于消息一般的一些处理，响应以及请求同用抽象父类中实现的那些方法；而对于响应以及请求消息处理的不同之处，两个实现类分别都继承了各自专用的接口（HttpResponse 以及 HttpRequest）。这样的设计使得代码既有很好的复用性，又能够方便解耦。做进一步的分析：无论是 Request 还是 Response 的独特处理部分出了问题，只需要关注其专用接口那部分的设计以及实现即可；而如果是对消息一般（共性）处理的代码要进行维护，则不需要关注具体的实现类，而在抽象类 AbstractHttpMessage 这个层次甚至更低的 HttpMessage 接口这一层次维护代码即可，而不需要去在更高的层次做一些修改。

根据之前的分析，header 在一则 message 中也起着至关重要的作用。接下来就来详细分析一下 header 以及维护消息头的 headergroup 这个容器的实现。

首先我们来介绍一下 Header 这个接口，它的实现十分简单：

```java
public interface Header extends NameValuePair {

    /**
     * Parses the value.
     *
     * @return an array of {@link HeaderElement} entries, may be empty, but is never {@code null}
     * @throws ParseException in case of a parsing error
     */
    HeaderElement[] getElements() throws ParseException;

}
```

图 5　Header 接口

可以看到它实际上是直接继承的 NameValuePair，这表明 Header 最基本的形式就是键值对了；此外还提供了一个解析消息头成分的方法。而 BasicHeader 就是 Httpcore 提供的最基本的一种消息头的实现类，它继承了 Header 这个接口，并且继承 Cloneable、Serializable 表示消息头是可拷贝，可序列化的。见下图：

```java
public class BasicHeader implements Header, Cloneable, Serializable {

    private static final HeaderElement[] EMPTY_HEADER_ELEMENTS = new HeaderElement[] {};

    private static final long serialVersionUID = -5427236326487562174L;

    private final String name;
    private final String value;

    public BasicHeader(final String name, final String value) {
        this.name = Args.notNull(name, "Name");
        this.value = value;
    }
    @Override
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
    @Override
    public HeaderElement[] getElements() throws ParseException {
        if (this.getValue() != null) {
            // result intentionally not cached, it's probably not used again
            return BasicHeaderValueParser.parseElements(this.getValue(), null);
        }
        return EMPTY_HEADER_ELEMENTS;
    }
    @Override
    public String getName() {
        return name;
    }
    @Override
    public String getValue() {
        return value;
    }
```

图 7　BasicHeader 实现类

BasicHeader 中重写了 getvalue、getname 这两个基本的 get 方法来得到键值对的值，此外还有 getelemets 方法解析消息头的成分以及 clone 方法实现消息头的复制等。除去最基本的 BasiceHeader 外，在 http/message 目录下还有一份 BufferedHeader.java 的代码，里面定义了 BufferedHeader 这个消息头的实现类，区别于 BasicHeader 的是 BufferedHeader 是已经格式化好的，从创建起就存于一个不可修改的 buffer 中，只有当它们的 value 被明确指定要解析时才会重新进行 parse 以及 format。BufferedHeader 类继承的接口不是 Header 而是专用的 FormattedHeader，这个接口本身继承了 Header 并提供了额外的一些操作。这再度体现了之前分析 Message 的各个接口时体现的层次关系：即从事物中提取共性，再根据特性实现层次化。

接下来分析一下 Headergroup 这个容器的实现：

```
/** The list of headers for this group, in the order in which they were added */
private final List<Header> headers;
```

图 8　Headergroup 实现类

它是通过一个 Header 的有序 List 来维护所有 headers 的，一些 clear、add、remove 的实现都可以直接调用 List 的方法；此外还有一些比较平凡的 get、set 方法就不在此一一列举，都是针对 header 来做的一系列操作。下面简要介绍一下我在阅读源码时着重思考的几个方法：

```
public void updateHeader(final Header header) {
    if (header == null) {
        return;
    }
    // HTTPCORE-361 : we don't use the for-each syntax, i.e.
    //     for (Header header : headers)
    // as that creates an Iterator that needs to be garbage-collected
    for (int i = 0; i < this.headers.size(); i++) {
        final Header current = this.headers.get(i);
        if (current.getName().equalsIgnoreCase(header.getName())) {
            this.headers.set(i, header);
            return;
        }
    }
    this.headers.add(header);
}
```

图 9　Headergroup 中更新消息头的方法

可以看出这个方法的参数为一个 header，实际做的事情是从 list 中找出与传入的 header 名字相同的第一个消息头并将它替换为新传入的 header；如果没有找到名字相同的则将其 add 进 list 中。之所以说这个方法让我比较感兴趣是因为，看到 update 这个方法名使我想起来之前阅读 Observor 设计模式里，观察者一个很典型的方法就是 update。Observor 设计模式的引入就是为了简化更新操作的流程并降低成本。当被观察的 Subject 更新时主动 notify 所有的 Obeservor，让它们自己调用 update 方法完成更新操作。观察者的观察的 Observor 要维护好观察者的名单，有相应的 attach 及 detach 方法。我起初觉得这里的设计是一种 Observor 设计模式的体现，headergroup 中这些操作消息头的方法可以为更高层的 HttpMessage 提供操作的接口，当在通信过程中捕捉到变化或被要求做出更新时，能通过这些方法迅速完成对 Message 的修改以及更新。但是后来我觉得这样是说不通的，看是否符合一个设计模式，首先应该分析有没有运用该设计模式的需求。在管理消息头的场景之中显然是没有这样的需求的，因为消息头在此处就应该是变动的源头，这里的 update 方法只是为管理消息头提供了操作的接口。如果 headergroup 这个容器类如果作为 Observor，我们甚至找不出它应该观察的 Subject，将其解释为一个观察者模式太过牵强。

```
public Header getCondensedHeader(final String name) {
    final Header[] hdrs = getHeaders(name);

    if (hdrs.length == 0) {
        return null;
    } else if (hdrs.length == 1) {
        return hdrs[0];
    } else {
        final CharArrayBuffer valueBuffer = new CharArrayBuffer(128);
        valueBuffer.append(hdrs[0].getValue());
        for (int i = 1; i < hdrs.length; i++) {
            valueBuffer.append(", ");
            valueBuffer.append(hdrs[i].getValue());
        }

        return new BasicHeader(name.toLowerCase(Locale.ROOT), valueBuffer.toString());
    }
}
```

图 10　　Headergroup 中得到压缩消息头的方法

getCondensedHeader 根据传入的 name 返回“压缩”的消息头，即它将所有同名的 header 的信息放在一个 header 里并返回；具体操作是将所有具有同样名称的消息头的 value 值拼接在一起得到 valuebuffer，再根据 name 以及 valuebuffer 新创建一个 header 对象并返回。

接下来就消息头容器对第三部分内容：高级设计意图进行简要分析。

三、设计模式探讨及高级设计意图分析

既然是一个消息头的容器，存储着各个消息头的信息，那么自然就会有遍历的操作。常规的思路有两种：一是将数据结构暴露出来，由外部提供遍历的方法进行操作；二是在容器内部提供一个遍历的方法，只向外部暴露这个遍历操作的接口。显然第二种方法要比第一种好一些，因为我们不愿意暴露数据的太多细节给外部，这样的做法往往面临着数据被修改的风险。但是提供一个接口的方法本身也不够好。一方面因为这样会使得一个容器承担着多重职责，除过提供操作接口、维护列表有序性外还要负责提供遍历方法，不同职责间存在关联且可能会冲突，而且使得管理的难度增大；另一方面，这样的迭代方法需要依赖于具体的数据结构，不符合面向对象编程中多态的设计思想。

进行了上述分析之后，我们来看一下在 httpcore 中是如何管理消息头容器的遍历操作的。然而最初在阅读代码的时候发现，headergroup 中并没有直接提供这样的遍历方法，而给出的是以下所示的两种 iterator 方法，它返回的是一个迭代器的对象。

```
public HeaderIterator iterator() {
    return new BasicListHeaderIterator(this.headers, null);
}

public HeaderIterator iterator(final String name) {
    return new BasicListHeaderIterator(this.headers, name);
}
```

图 11　　Headergroup 中的 iterator 方法

可以看到基于不同的请求（传参）方式，返回的都是一个 BasicListHeaderIterator 类

的实例化对象，只不过对象初始化的方法基于请求的不同而有所不同。一个用于遍历消息头容器中的所有 headers, 一个用于遍历消息头容器中所有名为 name 的同名 headers。那么为什么要通过一个对象来提供相应的遍历操作, 而不是在容器中直接进行遍历呢? 这就涉及到 JAVA 设计模式中非常重要的一个模式：**迭代器模式**。

首先我们指出引入迭代器模式遵循的最基本原则：单一职责原则。我们需要进行遍历操作的都是聚合对象，通俗来讲是一个容器。而容器，其最基本的职责就是通过一定的数据结构维护好存储的数据，保证它们的序关系，并对外部提供一些简单的操作接口来进行统一管理；至于遍历操作，在维护好存储对象的序列后并不是多么复杂的操作，不熟悉 java 设计原则的新手可能会觉得没有必要将其单独划分成另一种专门的职责，而是作为维护存储对象后一个附带的简单操作。但实际上，遍历这一职责显然是可分离的，而且也是可变化的（取决于访问的需求及方式）。无数 java 开发者们的经验指出，将职责明确划分开远比牵强附会地归为同一职责更加明智。

不妨说的更具体一点，我们来看第一个需求，在容器存储这些对象的时候容器不需要也不应该了解所存储对象的具体数据结构, 这是基于设计过程中要遵循的低耦合的设计原则。它只需要能够调用存储的单体对象的方法来完成相应设置，为外界管理提供操作接口即可。而牵扯到遍历的操作时，则一定会依赖于管理对象的具体数据结构，还会依赖于我们访问容器的具体请求方式。如果我们直接在容器之中实现遍历的具体方法，则我们访问容器对其中的数据进行遍历时：一会使得容器的设计及管理显得很复杂（因为必须要考虑具体的数据管理方式，还要考虑可能的多种遍历方式及访问需求）；二会使得容器的职责不明确（违背单一职责原则）；三会使得我们不可避免地要暴露容器的内部实现给外部，违背了低耦合的原则且可能对数据安全带来一定隐患。

而在我们采用迭代器设计模式后，实际上是将访问容器、进行遍历的职责分离了出来。这时，无论是 iterator 还是容器本身，它们的职责都单一而且明确。Iterator 大可以针对不同的遍历策略访问需求进行细化的设计，并且它只暴露给外界一个访问的光标，即指示当前访问到哪一个对象，而不会将容器内的内部细节暴露出来，增加了数据的安全性。

Iterator 设计模式往往和工厂方法模式结合在一起使用。因为在一个项目的设计之中，我们往往有多个不同的聚合对象即容器。对不同的容器，由于其管理的具体数据类

型以及用于管理的数据结构存在差异，我们往往有多个容器的具体实现类，而这些容器之间的共性都可以通过一个 Aggregete 的抽象基类提取出来。这个抽象基类就承担了工厂的角色，而具体的聚合对象诸如 httpcore 中的 headergroup 就是工厂生产出的一个实例。对于不同的聚合对象实体，我们要进行访问时的细节实现可能不同。如果我们采用迭代器模式对这些聚合对象进行访问，那么需要的迭代器也是各不相同的。基于以上需求，我们需要有一个通用的 iterator 接口提供基本的 hasNext()和 next()方法，以及若干具体的 iterator 实现类。这一点在 httpcore 中也有很好的体现：

```
public class BasicListHeaderIterator implements HeaderIterator {
    @Override
    public Header nextHeader()
        throws NoSuchElementException {

        final int current = this.currentIndex;
        if (current < 0) {
            throw new NoSuchElementException("Iteration already finished.");
        }

        this.lastIndex    = current;
        this.currentIndex = findNext(current);

        return this.allHeaders.get(current);
    }
```

图 12　　BasicListHeaderIterator 实现类

```
public class BasicHeaderIterator implements HeaderIterator {

    @Override
    public Header nextHeader()
        throws NoSuchElementException {

        final int current = this.currentIndex;
        if (current < 0) {
            throw new NoSuchElementException("Iteration already finished.");
        }

        this.currentIndex = findNext(current);

        return this.allHeaders[current];
    }
```

图 13　　BasicHeaderIterator 实现类

先关注 BasicHeaderIterator 以及 BasicListHeaderIterator 这两个实现类，它们继承了相同的接口 HeaderIterator，但是对于接口中的 nextHeader( )方法进行了不同的重写以符合各实现类的具体应用场景，这是工厂方法设计模式思想的体现。

接下来我们再来关注 BasicTokenIterator 以及 BasicHeaderIterator 这两个实现类：

```
public class BasicTokenIterator implements TokenIterator
public class BasicHeaderIterator implements HeaderIterator
```

图 14　　BasicListHeaderIterator、BasicTokenIterator 实现类比较

```
public interface TokenIterator extends Iterator<Object> {
    @Override
    boolean hasNext();
    String nextToken();
}
public interface HeaderIterator extends Iterator<Object> {
    @Override
    boolean hasNext();
    Header nextHeader();
}
```

图 15　　两个不同的接口

可以看到的是不同于上一页对比的两个实现类，此处对比的两个实现类分别实现了各自不同的专专用接口。而这两个专用接口本身又继承了统一的接口 Iterator，这是因为共性的存在，在继承相同接口的前提上，实现各自专有的方法又体现了差异性以及层次化的特征。这样的设计使得代码层次结构鲜明，同时又具有很好的复用性。

我们只需要在容器中调用 iterator 这个方法就能得到专用的迭代器，进而可以使用其中的遍历方法。而不需要基于每一种容器及其数据结构记住其专有的遍历方法，这其中所有的数据的细节都可以对用户隐藏；此外这种设计模式体现出了多态的思想，即调用公用的 iterator 接口可以返回不同具体形式的迭代器，使得代码灵活多变，也极大地方便了编程。

【参考文献】

【1】 https://www.cnblogs.com/SkyMouse/archive/2011/05/15/2340725.html

【2】 程杰.大话设计模式,2007(12).